REMARKS/ARGUMENTS

Claims 1-69 are pending in the present application. Claim 25 has been amended to correct a typographical error. Claims 1-69 remain pending. The Specification (Abstract) has been amended as requested by the Examiner.

The Examiner rejected claims 1-7, 15-18, 26, 36-38, 55-61 and 69 under 35 USC §103(a) as being unpatentable over Kuzara et al. (US 5,450,586) in view of Held (US5,889,988). The Examiner rejected claims 8-14, 19-25, 39-54, 62-68under 35 USC §103(a) as being unpatentable over Kuzara et al. (US 5,450,586) in view of Held (US5,889,988) and further in view of Cardoza et al. (US 5,630,049) Applicants respectfully disagree.

The present invention is a method and system for increasing software program security by obfuscation of program execution flow. The obfuscation of program execution flow hides key algorithms from view during debugging to increase the difficulty of reverse engineering, and to increase the difficulty of determining how to defeat anti-piracy features in the software. This is accomplished by hiding the execution flow of selected or critical portions of the program to be protected from a software debugger program when the debugger is run in an operating system that supports dual mode operation. i.e., user-mode (protected mode) and kernel-mode (unprotected mode).

In contrast, the references cited by the Examiner disclose methods for analyzing and debugging software. None of the references alone or in combination teach or suggest "executing the critical code segments within respective exception handlers, thereby *hiding execution of the critical code segments from a debugger program,*" as recited in claim 1. Instead, Kuzara discloses a system for debugging a software system in which code markers inserted by the user at compile time or interactively during a debug session <u>to make visible</u> critical points in code execution. (See Summary); and Held discloses a debugger that provides a system such that an end user can debug drivers of the operating system itself while still using a graphical user interface (Summary).

As stated on page 6, line 13+ of the Specification, a debugger is used to study the execution flow of a program, set software breakpoints, and analyze the contents of the microprocessor registers, data structures, and program variables at various points in the execution of a program. This process is normally associated with locating and correcting program errors during the development of a software program, and is referred to as "debugging" the software program. However, the same tool can be used to analyze the operation of any program, and thus is often used by hackers to determine the operation of a program they desire to make freely available. It can also be used to reverse-engineer software by helping an engineer extract key algorithms from a software program.

As state on page 9, line 21+, according to the present invention, the security of the software program is increased by obfuscating the execution flow of the program when the software program is executed on the computer system and analyzed by the debugger program. The execution flow of the software program is obfuscated by identifying critical code segments in the software program that need to be hidden, and then executing the non-critical portions of the software program in the user-level protected mode, as normal, while the critical code segments are executed within respective exception handlers.

User-level debuggers are incapable of analyzing code executed within exception handlers kernel-level mode and will therefore be unable to provide information regarding the critical code segments of the Software program for a hacker to reverse engineer. As the user issues commands to the debugger to study the program execution flow of software program, the user may potentially set software breakpoints to reduce the time required to analyze the program flow. As described above, however, there will be sections of the program flow that will be invisible to the debugger. It will not be apparent to the user that pieces of code are missing, or where these missing pieces are located, or even where they are positioned within the program execution flow. Rather, the flow of microprocessor instructions will seem to progress smoothly, but unexpected, seemingly uncomputed results will appear from nowhere. Even worse, key registers in the microprocessor will be seen to suddenly change values, altering the program flow in unexpected ways, for no apparent reason. Data structures will appear and disappear, or change their contents with no clear connection to the program execution flow. Finally, program execution flow may appear to randomly jump from place to place for no apparent reason. It is important to note that these inconsistencies of operation will not be readily apparent, and specifically at what points in the program execution flow the changes take place will not be readily apparent.

Not only do the references fails to teach or suggest the claims of the present invention, the references actually teach away from the claims of the present invention but because the references teach methods for making debugging code more visible, which makes debugging code easier, rather than hiding of execution of critical codes segments from the debugger.

The Examiner cited Col. 5, lines 30-49 of Kazura for teaching the hiding of execution of critical codes segments from a debugger program. However, this merely states:

> The first aspect of the present invention comprises means for inserting code markers directly into the embedded system under investigation at compile time or indirectly by inserting the code markers into the interface library of the RTOS service calls. As used herein, the RTOS service call library is the interface layer between the user's application code and the operating system "black box". By calling the RTOS service call library when calling the RTOS, these code markers may be inserted without the user knowing the inner workings of the operating system. This is accomplished by writing out information at all entry and exit points into and from the RTOS "black box", thereby making the RTOS activity visible without having to know the inner workings of the specific RTOS. The information is then written out to a specific, contiguous marker data area, and the written information identifies not only which RTOS service call was called but also the input parameters, the output parameters and the return values to/from the RTOS.

Thus, this passage teaches inserting code markers to make RTOS activity visible without the user having to know the inner workings of the RTOS, which is effectively hiding the operating system from the user, rather than hiding critical code segments from a debugger.

Held and Cardoza fail to cure this deficiency of Kazaur. Thus, Kazur in combination with Held and Cardoza fail to teach or suggest "executing the critical code segments within respective exception handlers, thereby *hiding execution of the critical code segments from a debugger program,*" as recited in claim 1. Thus, claim 1 is allowable over the references.

Independent claims 15, 16, 36, 55 and 69 recite similar and/or additional limitations and are also allowable over the references for at least the same reasons as claim 1. Moreover, the remaining claims, which depend either directly or indirectly from one of claims 1, 15, 16, 36, 55 and 69, are considered allowable for at least these same reasons.

In view of the foregoing, it is submitted that claims 1-69 are allowable over the cited references. Accordingly, Applicant respectfully requests reconsideration and passage to issue of claims 1-69 as now presented. Should any unresolved issues remain, the Examiner is invited to call Applicants' attorney at the telephone number indicated below.

Respectfully submitted,
STRATEGIC LAW GROUP

[illegible]

March 8, 2006
Date

Stephen G. Sullivan
Attorney for Applicant(s)
Reg. No. 38,329
(650) 969-7474